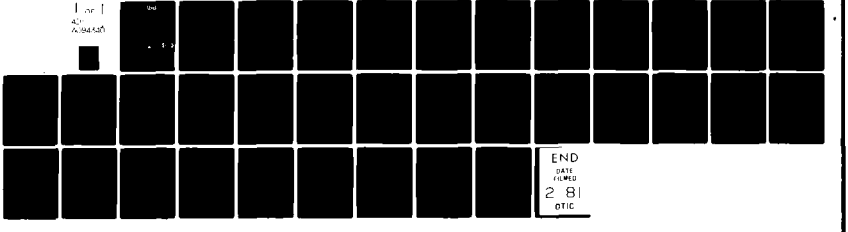
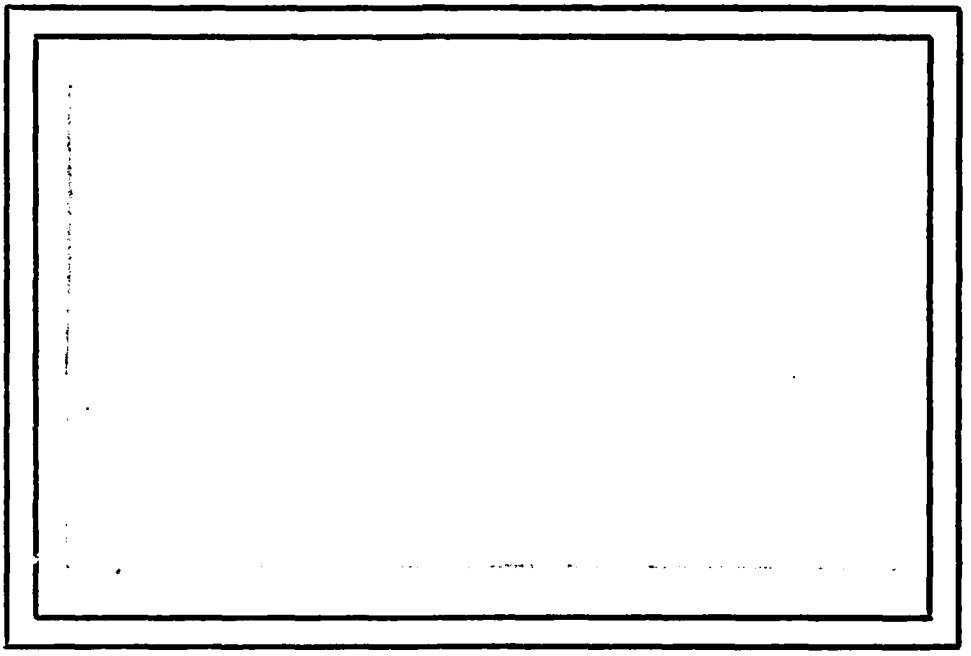
AD-A094 340 MARYLAND UNIV COLLEGE PARK COMPUTER VISION LAB F/G 9/2
PARALLEL COMPUTERS FOR REGION-LEVEL IMAGE PROCESSING.(U)
NOV 80 A ROSENFELD, A Y WU AFOSR-77-3271
UNCLASSIFIED TR-978 AFOSR-TR-81-0060 NL



AFOSR-TR- 81-0060

LEVEL II (12)



COMPUTER SCIENCE
TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: AFOSR-TR-81-0060	2. GOVT ACCESSION NO.: AD-A094340
	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): PARALLEL COMPUTERS FOR REGION-LEVEL IMAGE PROCESSING.	5. TYPE OF REPORT & PERIOD COVERED: INTERIM rept.
	6. PERFORMING ORG. REPORT NUMBER: TR-978
7. AUTHOR(s): Azriel Rosenfeld, Angela Y. Wu	8. CONTRACT OR GRANT NUMBER(s): AFOSR-77-3271
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Computer Vision Laboratory, Computer Science Center, University of Maryland, College Park, MD 20742	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61102F 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS: Math. & Info. Sciences, AFOSR/NM, Bolling AFB, Washington, DC 20332	12. REPORT DATE: November 1980
	13. NUMBER OF PAGES: 52
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Pattern recognition
Image processing
Parallel processing

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

It is well known that parallel computers can be used very effectively for image processing at the pixel level, by assigning a processor to each pixel or block of pixels, and passing information as necessary between processors whose blocks are adjacent. This paper discusses the use of parallel computers for processing images at the region level, assigning a processor to each region and passing information between processors whose regions are related. The basic difference between the pixel and region

UNCLASSIFIED

levels is that the regions (e.g., obtained by segmenting the given image) and relationships differ from image to image, and even for a given image, they do not remain fixed during processing. Thus one cannot use the standard type of cellular parallelism, in which the set of processors and interprocessor connections remain fixed, for processing at the region level. Reconfigurable cellular computers, in which the set of processors that each processor can communicate with can change during a computation, are more appropriate. A class of such computers is described, and general examples are given illustrating how such a computer could initially configure itself to represent a given decomposition of an image into regions, and dynamically reconfigure itself, in parallel, as regions merge or split.

TR-978
AFOSR-77-3271

November 1980

PARALLEL COMPUTERS FOR REGION-LEVEL IMAGE PROCESSING

Azriel Rosenfeld

Angela Y. Wu*

Computer Vision Laboratory
Computer Science Center
University of Maryland
College Park, MD 20742

ABSTRACT

It is well known that parallel computers can be used very effectively for image processing at the pixel level, by assigning a processor to each pixel or block of pixels, and passing information as necessary between processors whose blocks are adjacent. This paper discusses the use of parallel computers for processing images at the region level, assigning a processor to each region and passing information between processors whose regions are related. The basic difference between the pixel and region levels is that the regions (e.g., obtained by segmenting the given image) and relationships differ from image to image, and even for a given image, they do not remain fixed during processing. Thus one cannot use the standard type of cellular parallelism, in which the set of processors and interprocessor connections remain fixed, for processing at the region level. Reconfigurable cellular computers, in which the set of processors that each processor can communicate with can change during a computation, are more appropriate. A class of such computers is described, and general examples are given illustrating how such a computer could initially configure itself to represent a given decomposition of an image into regions, and dynamically reconfigure itself, in parallel, as regions merge or split.

The support of the U.S. Air Force Office of Scientific Research under Grant AFOSR-77-3271 is gratefully acknowledged, as is the help of Janet Salzman in preparing this paper.

*Also with the Dept. of Mathematics, Statistics, and Computer Science, American University, Washington, DC.

1. Introduction

The use of parallel, "cellular" computers for image processing at the pixel level was proposed over 20 years ago [1]. In this approach, the image is divided into blocks of pixels (or even into single pixels, if enough processors are available), and a processor is assigned to each block. Local operations on the image can then be performed very efficiently in parallel, with information being passed as necessary between processors whose blocks are adjacent. (Parallelism can also be used in other types of operations, such as discrete transforms or geometric operations; but local operations are probably the most common.) Computers embodying this approach, using many thousands of processors, are gradually beginning to appear [2-5].

Pixel-level processing is the primary type of computation employed in image-to-image operations ("image processing" in the strict sense), and is also heavily used in the early stages of image analysis (also known as scene analysis, "image understanding", computer vision, or pictorial pattern recognition), where the goal is a description of the image, not a processed version of it. The cellular computer architecture is very well suited for such computations, since the images being processed are usually all of the same size, and the array structure of an image remains fixed throughout processing (except possibly for some losses of information at the

borders). Thus, pixel-level image processing can be very efficiently performed by a cellular computer in which the interprocessor connections constitute a hardwired array structure.

The situation at the later stages of image analysis is rather different. Here the image is segmented into regions, and additional processing takes place at the region level, rather than the pixel level. Typically, the regions are represented by the nodes of a graph or similar data structure, and relationships between regions (e.g., adjacency) are represented by the arcs of the graph. Processing at the region level might then involve merging or splitting of regions, or matching configurations of regions between two images or with models. Some general examples of region-level processing will be given in Section 4; for an extensive treatment of the subject, see [6].

Since region-level processing can involve combinatorial searches of graphs, it can be somewhat time consuming; the number of regions is much smaller than the original number of pixels, but it is still non-negligible (e.g., it may be several hundred). Thus the use of parallelism is desirable even at the region level. One way of achieving parallelism is to use a graph-structured cellular computer or "cellular graph", [7] in which a processor is assigned to each node (= region), and processors corresponding to related nodes are connected to each other. Note that the number of processors that would typically be required, e.g. several hundred, is easily achievable at reasonable cost.

An important difference between the pixel and region levels of processing is that the graph structures arising at the region level vary from image to image, and even for a single image, they do not remain fixed during processing (e.g., regions may merge or split). Thus one cannot use a fixed-structure cellular graph; rather, some type of reconfigurable cellular graph is needed. Instead of processors that are hardwired into a particular graph structure, one should think in terms of processors that have lists of "addressees" with whom they can communicate, and where these lists can change in the course of a computation. On such "reconfigurable cellular computers" see [8-10].

Section 2 of this paper reviews the concept of a reconfigurable cellular computer; it discusses in particular why it may be desirable to keep the size of the address lists bounded, and what this restriction implies about the nature of the reconfiguration process. Section 3 reviews various types of region-level image descriptions, such as adjacency graphs, boundary segment graphs, and quadtrees, and shows how such graphs could be created by a reconfiguration process starting from an array-structured cellular processor in which the given image is initially stored. Section 4 illustrates how a reconfigurable cellular computer would be used in region-level image analysis, particularly for such tasks as region merging and matching of region configurations.

2. Reconfigurable cellular computers

Abstractly, a cellular computer can be modeled by a graph whose set of nodes is the set of processors, and where two nodes are joined by an arc iff the corresponding processors can communicate directly. We will assume that communication is two-way, so that the graph is undirected. We will also assume that the graph is connected.

In a non-reconfigurable cellular computer [7], the graph structure remains fixed throughout a computation; thus we can regard those pairs of nodes that are joined by arcs as being hardwired together. In the reconfigurable case [8-10], on the other hand, the graph structure may change during a computation, and it is more appropriate to think of each processor as having an "addressee list" specifying those processors with which it can currently communicate. The physical realization of the communication process will not be discussed here, but it should be pointed out that the ZMOB system [11], a collection of 256 microprocessors that communicate via a fast bus, can simulate a reconfigurable cellular computer having up to 256 nodes.

In [10] it is argued that it may be useful to require the addressee lists of the processors in a reconfigurable cellular processor always to remain small, and in any case of bounded size. This helps to prevent the amounts of I/O that the processors must do at a given stage of the

computation from becoming extremely unequal, which would tend to impair the efficient parallel use of the processors. The use of bounded addressee lists, say all of (about) the same length ℓ, also makes it reasonably efficient to implement the communication process by means of a shift-register bus, as in [11]. For example, suppose that the computation is divided into stages, during each of which each processor does approximately the same amount of computation. At the end of each stage, each processor then sends (and receives) ℓ messages via the bus; delivery of all these messages (assuming them to be of the same size) is completed in ℓ "turns" of the bus, and the next computation stage can begin. Thus by using addressee lists that are all of the same size, we can maintain a high degree of parallelism in the interprocessor communication process.

Of course, the requirement of bounded addressee lists is not appropriate for all computational tasks. However, as we shall see in Sections 3 and 4, it is appropriate in many types of region-level image analysis, where the computational tasks can in fact be divided evenly among the processors and broken up into stages, and where the interprocessor communications required at each stage involve the passing of information from each processor to a small set of the others.

It should be realized that when we limit the addressee lists to bounded size, irrespective of the total number of processors, we are making it impossible for most pairs of

processors to communicate directly. If it is necessary for two arbitrary processors to communicate, they must do so by passing messages through a sequence of intermediate processors, and the number of such "relay" stages required can be as high as the diameter of the graph [10]. Evidently, we should not insist that the lists be of bounded size unless the number of processors with which a given processor will need to communicate directly does in fact remain bounded at every stage of the computation. As we shall see in Sections 3 and 4, this is indeed often true.

Another problem with bounded addressee lists is how to insure that they *remain* bounded when they are being changed. If two processors can currently address one another, it is easy for them to agree to drop one another from their lists and then to do so simultaneously. But if they cannot address one another, it is less obvious how to get them to add each other to their lists. The approach to this problem discussed in [8-10] is as follows: If processors A and C have a common neighbor B, and A wants to add C to its list, A informs B; B checks with C; and if they agree, B tells them to add each other simultaneously (and to drop B, if desired). If they do not have a common neighbor, say C is at distance ℓ from A in the graph, A can use the scheme just desribed to add to its list a processor at distance $\ell-2$ from C, so that A's distance from C is now $\ell-1$; by repeating this process,

A can thus eventually add C to its list. As we shall see in Section 4, list changes often do involve pairs of processors that have common neighbors, so that these changes can in fact be carried out quite efficiently in parallel.

In summary, the model of computation that will be used in this paper assumes a collection of processors whose interconnections, at any given stage, define a graph of bounded degree - i.e., each processor can communicate directly with a small set of others, defined by its addressee list. We further assume that additions to the addressee lists are carried out stepwise, by way of common neighbors, as described above. We will now show how this model can be used to set up interconnection structures representing the region relationships in a given region-level image description, and to modify these structures in the course of region-level processing.

3. Building region-level representations

Suppose an image has been input to an array-structured cellular computer, and as a result of processing at the pixel level, the image has been segmented into regions. We assume that the regions are identified by labels, i.e., that a unique label is associated with each region, and every pixel in that region has been given that label.

We now want to process the image at the region level, e.g., to carry out region merging operations, or to find configurations of regions that match a given configuration. These types of processing no longer need to make use of the pixel array; they are more efficiently carried out on a graph structure in which, e.g., the nodes represent regions, labeled with the values of various region properties that will be used in the processing, and the arcs represent relationships between pairs of regions. Thus such region-level processing can be implemented in parallel on a cellular computer having the given graph structure, as we will see in the next section.

In this section, we show how to "construct" cellular computers having various useful types of graph structures by reconfiguring the original array structure. The graph structures for which we do this include the region adjacency graph and the region boundary segment graph (see below for their precise definitions); it should be pointed out that the adjacency graph need not have bounded degree. We also show

how to construct cellular computers in which the nodes do not correspond to regions, but rather to "maximal blocks" belonging to the regions, such as those used in run length and quadtree regions representations; these structures too can be used for efficient processing of region information.

3.1 The adjacency graph

Given a segmentation of an image into connected regions, the adjacency graph is defined as having nodes corresponding to the regions, with two nodes joined by an arc iff the corresponding regions are adjacent. Note that some nodes in this graph may have very high degree, on the order of the image area (consider a large region that has many tiny holes); thus our preferred assumption of bounded degree cannot be assured. In spite of this, we treat the case of the adjacency graph here because it is such a basic relational region representation.

We now describe how to construct a graph-structured cellular computer having the adjacency graph of the given segmentation as its graph structure. A simple method of doing this would be as follows: We assign the labels of the regions as "addresses" to a new set of processors, one per label. We also determine that the set of adjacent pairs (i,j) of region labels, i.e. pairs such that a pixel having label i is adjacent to a pixel having label j, and we give the processor having label i the address list $\{j \mid (i,j) \text{ is a pair}\}$. The problem with this simple approach is that to extract the set of label pairs (and for that matter, the set of labels) from the array of processors containing the segmented image, we must read them out in sequence; this requires an amount of time proportional to the image diameter (the labels or pairs must be shifted out of the array) and to the number of labels and pairs. In the following paragraphs we describe a more complicated method of constructing the desired

adjacency-graph-structured cellular computer by reconfiguring the given array of processors itself. The process is carried out for all the regions simultaneously, and for each region, it takes time proportional to the region's perimeter or intrinsic diameter.

We first identify a unique processor associated with each region - e.g., its "northwest corner" (i.e., the processor containing the leftmost of the uppermost pixels in the region). This can be done as follows: Each processor whose north and west neighbors are not in the region sends a message, containing its coordinates, around the region border on which it lies. If the message reaches a processor having a higher y coordinate, or the same y coordinate and a lower x coordinate, it is erased. Evidently, the only message that can get all the way around its border without being erased is the one that originated at the true northwest corner, which is on the outer border. The time required for this process is proportional to the (outer) perimeter of the region. Note that the process can be carried out for all regions simultaneously.*

*A somewhat more complicated process, described in [2], uses messages whose size remains fixed no matter how large the array, rather than messages that contain coordinates. If we use messages that need not stay inside the given region, we can identify the northwest corner of a region in time proportional to the diameter of the region's circumscribed rectangle, which may be much smaller than the region's perimeter. However, it is hard to do this efficiently for all regions at once, since many messages may have to pass through a given pixel simultaneously.

Now that we have identified a unique "corner" processor in each region, we can use it as the region's representative in the adjacency-graph-structured computer; but we need to give it the addresses of the processors representing the regions adjacent to the given one. We can "broadcast" their addresses to it as follows: Each corner processor "propagates" its address throughout its region (passing it as a message from processor to processor as long as they have the same label), and also propagates it into the adjacent regions, marking it as soon as it crosses the region border so that it cannot cross a border again. The addresses that reach a corner processor as a result of this propagation are just the addresses of the corner processors of the adjacent regions. The propagation takes time proportional to the sum of the two regions' intrinsic diameters. When messages meet one another in the course of the propagation process, they can be sorted (e.g., in lexicographic order) and duplicates discarded; but a processor may still have to transmit as many messages as there are regions adjacent to its region, so that the propagation time is also proportional to this number.

In the address-passing scheme just described, we have not used the concept of local exchange of addresses, but have simply broadcast them across the regions. In the following paragraphs we describe a more complicated scheme which does

use local exchange, and which also uses a spanning tree for each region, and a border-following process to initiate the messages, in order to greatly reduce the number of messages that are initiated, and thus reduce the amount of sorting involved in passing them.

We begin by constructing a spanning tree for each region, rooted at its "corner" processor. To this end, the corner broadcasts a signal throughout its region, and when each processor receives the signal, it notes the neighbor from which it was received, resolving ties according to some specified rule. It is easily seen that this process defines a spanning tree for the region, where the "father" of each processor is the unique neighbor (or tie-winner) from which it received the signal. Such trees are constructed for all the regions simultaneously. The time required for this construction is proportional to the intrinsic diameter of the region.

To exchange addresses between adjacent regions, the northwest corner N of each region R passes its address (i.e., its coordinates) around the outer border B of R on which it lies - i.e., N exchanges addresses with a neighbor along the border, then with a neighbor of that neighbor (dropping the original neighbor), and so on. As each processor along B receives the address, it checks its neighbors that do not lie in R. As we move around B, these neighbors lie in a sequence

of regions adjacent to R. Each time we come to a new such region, say at neighbor N_i, N exchanges addresses with N_i (and does not drop it). Thus, when we have gone completely around B, N has exchanged adresses with a pixel N_i in each region R_i that is adjacent to R along B (note that some of those R_i's may be the same, since a region may touch R in several places). At the same time, we can begin to pass these addresses up the spanning trees of the R_i's - i.e., N exchanges addresses with N_i's father (and drops N_i), then with the father's father, and so on until the root of R_i's spanning tree is reached. Note that if there are several N_i's in the same region, we will reach the root along several different paths; the duplicate addresses can be discarded.

As a result of the process just described, each N has exchanged addresses with the northwest corners of all the regions that meet R along its outer border B. Let R' be a region that meets R along a hole border of R; this implies that R meets R' along the outer border of R', so that N has also exchanged addresses with the northwest corner of R'. Thus when this process has been carried out (in parallel!) for every N, addresses have been exchanged between every pair of northwest corners whose regions are adjacent, so the desired adjacency graph has been constructed. Note that if R and R' are both adjacent along their outer borders, they will exchange addresses at least twice, but we can discard the duplicates.

While constructing the adjacency graph, it is straightforward to compute various properties of each region R and store them in N. For example, to computer the area of R, each node of the spanning tree adds the numbers computed by its sons (or counts itself as 1, if it is a leaf node) and passes them up the tree to its father; when this process is complete, the root (i.e., N) has computed the total number of nodes, i.e., the area. Similarly, N can compute the sum of the gray levels of the pixels in R, and divide it by the area to obtain the mean gray level. As another example, N can compute the perimeter of R, defined as the number of border pixels, by having each border pixel mark itself and then counting only the marked pixels.

Property values associated with pairs of regions can also be computed and stored; for example, N can compute the length of border which its region R has in common with each of the adjacent regions R', and store that value together with the address of the northwest corner of R'. To do this, we modify the address - passing scheme described above to make N_i the last neighbor along each border segment, rather than the first. It is then easy to keep a count of the border pixels along that segment (i.e., since the previous N_i was found), and let both N and N_i store that count when they exchange addresses. If R' is adjacent to R along several segments, all of their counts will reach the root of the

spanning tree of R', where they can be summed to obtain the total count. Similarly, N can compute the sum (and hence the average) of the absolute gray level differences around its borders ("border strength").

3.2 The border segment graph

In this subsection we show how to construct a more complicated type of graph in which the nodes represent border segments along which pairs of regions meet, and pairs of nodes representing consecutive or adjacent border segments are joined by arcs. We will regard a border as being composed of "cracks" between adjacent pairs of pixels, so that the border belongs to neither of the two regions. Of course, each "crack" is in fact represented by the processor associated with one of the two pixels meeting at it (e.g., the one above a horizontal crack, or to the left of a vertical crack); but to simplify the discussion, we will regard the cracks as having processors associated with them directly. Since a crack is adjacent to only three other cracks at each end, a border segment cannot be consecutive with more than six other border segments, so that the graph of border segment adjacencies always has bounded degree, unlike the region adjacency graph.

Along each border of a region R, the region is adjacent to a sequence of (one or more) other regions. If R is adjacent to only one other region, we can regard, e.g., the "northwest corner" horizontal crack on the border, identified as in Section 3.1, as the "head crack" of that border with respect to both R and the other region. If R is adjacent to two or more regions, along a sequence of border segments, we regard

the first crack along each segment (say in the sequence for which R is on the right) as the head crack of that segment with respect to R; and similarly, the last crack becomes the head crack with respect to the other region. Each head crack can identify itself in a bounded amount of time by locally examining its neighbors along the border. We then pass its address around the border so that it shares addresses with the head cracks of the preceding and following segments. Since the head crack of a segment is at a border branch point, i.e., a place where three or four regions meet, it can also share addresses with the head cracks of the other border segments that branch from it. This process is carried out in parallel for all the branch points; it takes time proportional to the maximal branch length. Note that it is much simpler and faster than the process of constructing the region adjacency graph, but of course it yields a structure that has many more nodes.

We can label the head crack of each border segment with the ordered pair of labels of the regions that meet at the crack, and can also store at it the length of the segment (or even its "crack code", if space permits). If we want to associate a unique processor with each region border, we can identify the "northwest corner" horizontal crack of the border, as above, and distinctively mark the head crack of the border segment that contains it. Since we know which is the

outer border of each region (it is the one for which the region is on the right when we follow the border clockwise), we can use a special distinctive mark for that border; this associates a unique processor with each region. This process, carried out in parallel for each region, takes time proportional to the (outer) perimeter of the region. Note, however, that this structure does not provide connections between the processors that represent adjacent regions, or even between the processors that represent different borders of the same region, since these borders do not meet.

3.3 The run graph

Each row of a segmented image consists of successive runs (= maximal sequences) of pixels belonging to the various regions. A run on a given row is preceded and followed by runs belonging to different regions on its own row (except at the ends of the row), and is also adjacent to one or more runs belonging to the same or different regions on the adjacent rows. The run graph has nodes corresponding to the runs and arcs corresponding to pairs of adjacent runs. Note that it need not have bounded degree; a run can be adjacent to a number of other runs proportional to its length.

Construction of the run graph is quite straightforward. Each left run end identifies itself, and sends messages leftward along the run on its left, and rightward along its own run, to exchange addresses with the left ends of the runs preceding and following it. As the rightward-moving message passes left run ends on the rows above and below, addresses are also exchanged with these. Messages are also sent leftward on the rows above and below (starting from just above and below the left end) to find the left ends of the leftmost runs adjacent to the given run on these rows. The time required by this process, which is carried out in parallel for all the runs, is proportional to the maximum run length. Of course, the left end can also store the length of its run, as well as its label.

In the run-graph-structured cellular computer representing a segmented image, the northwest run of each region can identify itself, and the northwest runs of adjacent regions can exchange addresses, using procedures similar to those described in Section 3.1. Similarly, the northwest run of each region can store various properties of the region, much as in Section 3.1. The propagation process involved may be somewhat faster, since messages pass from run to run rather than from pixel to pixel, but in the worst case, they still take time proportional to the region diameters.

3.4 The quadtree

A segmented image of size 2^n by 2^n can be decomposed into blocks, each entirely contained within a single region, by the following recursive subdivision procedure: If the region consists of a single region, we are done; if not, we decompose it into quadrants and repeat the process for each quadrant; and so on. We can represent the results of this procedure by a tree of degree 4 (a "quadtree") as follows: The root of the tree represents the whole image; whenever we split a block into quadrants, we give that block's tree node four sons; when we do not split a block, we give its node the label of the region to which the block belongs. This tree is called the quadtree of the given segmented image; note that it is a graph of bounded degree (≤ 5).

We now briefly describe how to construct a quadtree-structured computer corresponding to (the quadtree of) a given segmented image; for further details see [12]. The center row and column (e.g., rounded down) of the image are marked, and the center pixel identifies itself as the root node of the tree. The pixels in each quadrant propagate their labels to the center pixel (labels shift in the appropriate two directions, and are stopped by the marked row and column). If it receives only one label from each quadrant, and these labels are all the same, the tree construction is finished. Otherwise, the center row, column and pixel of each quadrant identify themselves, and the center pixel exchanges addresses with the center pixel of the image, so that

it becomes one of the root node's sons. The process is now repeated in parallel for each quadrant that needs to be subdivided (not all its pixels have the same label). The propagation processes at the k^{th} stage take time proportional to 2^{n-k}, so that the total propagation time is proportional to $2^n + 2^{n-1} + \ldots \simeq 2^n$, the image diameter.

If desired, in the process of constructing the quadtree, we can establish links between nodes whose blocks are spatially adjacent. We do this by linking each center pixel to the center pixels of its neighboring quadrants; if a quadrant is found to have more than one label, these links are passed to its appropriate sons (e.g., the link to its east neighbor is passed to its northeast and southeast sons). If the northwest corner of each region has identified itself, the quadtree node whose block contains that corner can also identify itself, and that node can compute and store various properties of the region, as before.

4. Region-level processing

4.1 Region merging

Many standard region merging procedures can be carried out using a graph representation of the segmented image; merging decisions are made on the basis of the properties stored at adjacent pairs of nodes, and when a pair is merged, the properties of the new node are computed from the properties of the pair, without the need to refer back to the pixel data. Thus if the graph representation is embodied in a cellular computer, merging decisions and property updating can be computed in parallel. When region merging is controlled by a model for the types of regions expected in the image, computation of merge merits may be a major task, involving a variety of conditional probability computations, and it becomes especially important to carry out this task in parallel. Note that when merges are made in parallel, one should not merge a pair of nodes unless they have mutually chosen to merge with each other; otherwise, node A might merge with B, and at the same time B might merge with C (etc.).

As an example of how region merging might be performed, consider the adjacency graph of the regions, and suppose that each node has stored the area, perimeter, and average gray level of its region, as well as the length and strength of its common border with each adjacent region. We can thus compute, for each pair of neighboring regions, a merge cost based on the difference between their average gray levels,

on the strength of their common border, and on the ratio of their common border to total perimeter [13]. If two regions, say R and R', have mutually lowest merge costs, we merge them by picking one of them, say R, to represent the merged region. The neighbors of R' then exchange addresses with R, and R' is dropped. At the same time, the properties of the merged region are computed from those of R and R': the areas add; the average gray levels are averaged, with weights proportional to the areas; the perimeters are added and the length of common border is subtracted from the sum. This is done in parallel for all pairs of mutually best neighbors. It takes O(constant) time, since it involves exchanges of information only between neighbors, and does not require propagation of information around the graph.

Region merging can also be carried out efficiently using the border segment graph. If the two region meet only once, their common border segment is dropped, and the remaining parts of that border for each of them are linked together into a single border. If they meet more than once, two or more borders get created out of the remaining parts; the details will not be given here.

Another important example of region merging involves quadtree approximations to an image. Given an image, we measure its inhomogeneity, e.g., its gray level standard deviation;

if this is higher than some threshold, we divide the image into quadrants and repeat the process for each quadrant. The result of this recursive subdivision process is a decomposition of the image into homogeneous blocks, which can be represented by a quadtree. Cellular computers embodying such quadtrees can be constructed as described in Section 3.4, except that instead of checking that all labels in a block are the same, we compute the block's standard deviation and check that it is below threshold. (The block computes its mean by summing its gray levels and dividing by its area; subtracts this mean from each gray level, squares the results, sums them, and takes the square root - all in time proportional to the block diameter.)

Adjacent blocks in a quadtree may be very similar or even identical, so that even if they are merged, the standard deviation remains below threshold. We can test pairs of adjacent blocks (which may be assumed to be linked, as indicated at the end of Section 3.4) and merge them if they satisfy this criterion, using a suitable priority ordering to insure that a block does not attempt to merge with two of its neighbors. Note that the mean of the union is the average of the means, weighted by the block areas; and the variance of the union can similarly be computed from the means and variances of the blocks. This process can be repeated to yield a final set of regions for which no further merging is possible. Note that this final set depends on the sequence

in which the merging is done (i.e., on the priorities). Of course, as soon as we start merging adjacent blocks, the result is no longer a quadtree; the merging is done on the adjacency graph of the blocks (which we have assumed to be linked), and the quadtree links are no longer used.

Parallel merging provides an alternative approach to constructing the adjacency graph of a segmented image, starting from the array of processors. By repeated merging of adjacent pairs of nodes that have the same label, we can reduce each region to a single node, which is linked to the nodes representing the adjacent regions. Note that during the merging process, a node may get linked to a very large number of other nodes by inheriting the neighbor relationships of the nodes that have been merged with it. Note also that we cannot do the merging for all pairs of nodes simultaneously, since a node must not merge with more than one other node at a time, even though it belongs to several pairs; and we must define criteria for deciding, when two nodes merge, which of them is discarded. As a simple example of how to handle this, suppose that we merge nodes with their north neighbors (or with the westmost of their north neighbors, if they have several as a result of previous merging), provided the latter are north border nodes (i.e., have no neighbors on the north with the same label), and with their west neighbors provided they are west border nodes, with north having priority over west if both possibilities exist. If

this is done repeatedly, the surviving node of each region will be its northwest corner, and the time required will be proportional to the intrinsic diameter of the region.

Merging can also be used to construct the adjacency graph starting from the run graph or quadtree; e.g., we can merge runs or quadtree blocks with their north or west neighbors (recall that in a quadtree we can link pairs of nodes whose blocks are neighbors) as indicated in the preceding paragraph.

4.2 "Symbolic matching"

Suppose we are given a graph representation of a segmented image, with property values associated with the nodes, and we want to detect the presence of configurations of the regions that match a given configuration, which we assume to be represented by a graph of the same type. If the image graph is embodied in a cellular computer, each node can check its neighborhood to determine whether the given configuration is present. For arbitrary graphs, if this is done in parallel by all nodes, the checking processes may interfere with one another; but we can initially process the graph so as to insure that this will not happen, e.g., by coloring the graph so that no two nodes have the same color if their distance apart is less than the diameter of the configuration [7]. Once this coloring has been done, the checking time is proportional to the diameter of the configuration.

Rather than requiring exact matches, we can compute mismatch measures between the configuration and the subgraphs of the given graph, and look for below-threshold mismatches; this too can be done in parallel by all nodes. We can also use a "relaxation" process, applied in parallel at all nodes, to eliminate nodes from consideration as match possibilities if they do not have the proper sets of neighbors, or to reduce their potential match scores if their neighbors do not have property and relationship values close to the desired ones [14-15].

5. Concluding remarks

Graph representations of the regions in an image contain much less information than the original image, but it still may be desirable to process them in parallel, e.g., in real time situations. This can be done efficiently using graph-structured cellular computers embodying the given graph representation. Such computers typically require only a few hundred processors and so can be built today at reasonable cost. This paper has described how such computers can be configured, starting from the array of processors containing the segmented image. It has also discussed how they can be used to carry out various types of region merging and graph matching tasks. As hardware realizations of graph-structured cellular computers begin to emerge, they should find many practical applications in real-time region-level image processing and analysis.

References

1. S.H. Unger, A computer oriented toward spatial problems. Proc. IRE 46, 1958, 1744-1750.

2. B.H. McCormick, The Illinois pattern recognition computer - ILLIAC III, IEEE Trans. EC-12, 1963, 791-813.

3. M.J.B. Duff and D.J. Watson, The cellular logic array processor, Computer J. 20, 1977, 68-72.

4. K.E. Batcher, Design of a massively parallel processor, IEEE Trans. C-29, 1980, 836-840.

5. P. Marks, Low level vision using an array processor, Computer Graphics Image Processing, 1980, in press.

6. T. Pavlidis, Structural Pattern Recognition, Springer, New York, 1977.

7. A. Wu and A. Rosenfeld, Cellular graph automata (I and II), Info. Control 42, 1979, 305-329, 330-353.

8. A. Wu and A. Rosenfeld, Local reconfiguration of networks of processors, TR-730, Computer Vision Laboratory, Computer Science Center, University of Maryland, College Park, MD, February 1979.

9. T. Dubitzki, A. Wu, and A. Rosenfeld, Local reconfiguration of networks of processors: arrays, trees, and graphs, TR-790, Computer Vision Laboratory, Computer Science Center, University of Maryland, College Park, MD, July 1979.

10. A. Rosenfeld and A. Wu, Reconfigurable cellular computers, TR-963, Computer Vision Laboratory, Computer Science Center, University of Maryland, College Park, MD, November 1980.

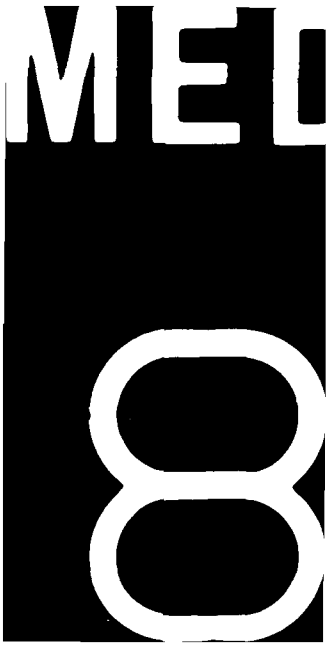
11. C.J. Rieger, ZMOB: A mob of 256 cooperative Z80A-based microcomputers, Proc. DARPA Image Understanding Workshop, November 1979, 25-30.

12. T. Dubitzki, A. Wu, and A. Rosenfeld, Region property computation by active quadtree networks, TR-823, Computer Vision Laboratory, Computer Science Center, University of Maryland, College Park, MD, November 1979.

13. C.R. Brice and C.L. Fennema, Scene analysis using regions, Artificial Intelligence 1, 1970, 205-226.

14. L. Kitchen and A. Rosenfeld, Discrete relaxation for matching relational structures, IEEE Trans. SMC-9, 1979, 869-874.

15. L. Kitchen, Relaxation applied to matching quantitative relational structures, IEEE Trans. SMC-10, 1980, 96-101.


MEL
8